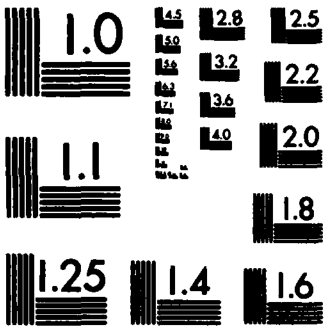
AD-A134 003 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR ADA INTEGRATED ENVIRONMENT..(U) INTERMETRICS INC CAMBRIDGE MA 22 MAR 83 IR-681-1 F30602-80-C-0291 1/1

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(1)

AD-A134003

CONTRACT F30602-80-C-0291

IR 681-1 COMPUTER PROGRAM DEVELOPMENT SPECIFICATION FOR Ada INTEGRATED ENVIRONMENT: PROGRAM INTEGRATION FACILITIES TYPE B5

B5-AIE (1).PIF (1)

22 MARCH 1983

PREPARED FOR: ROME AIR DEVELOPMENT CENTER
CONTRACTING DIVISION/PKRD
GRIFFISS AFB, N.Y. 13441

PREPARED BY: INTERMETRICS, INC.
733 CONCORD AVE.
CAMBRIDGE, MA 02138

This document was produced under Contract F30602-80-C-0291 for the Rome Air Development Center. Mr. Donald Mark is the Program Engineer for the Air Force. Mr. Mike Ryer is the Project Manager for Intermetrics.

CONTENTS

FIGURES

1. Scope

1.1 Identification

Interface

This document establishes the performance, design, test, and qualification requirements for the Program Integration Facilities ~~(PIF)~~ for the Ada Integrated Environment. These facilities include the Program Library ~~Intpface~~ Packages, the Program Builder and the Program Library Support Tools. This document also includes requirements for the design of the Ada program library, the environment within which program integration occurs.

The CPCI's that comprise the PIF are listed below, along with their component CPC's.

CPCI	CPC
Program Library Interface Packages (PLIF)	
	Library Object Identification (A)
	Library Object Dependency Manager (B)
	Library Object Allocation Package (C)
	Library Configuration Management (D)
Program Builder (PBUILD)	
	Builder (A)
	Preamble Generator (B)
	Program Completeness Checker (C)
	Body Generator (D)
	Linker (E)
Program Library Tools (PLTOOLS)	
	Program Library Manager (A)
	Change Analyzer (B)
	Recompilation Minimizer (C)
	Link Map/X-Ref Lister (D)
	Source Reconstructor (E)
	Unit Lister (F)
	Foreign Object Module Importer (G)

1.2 Functional Summary

The Program Integration Facilities provide a complete set of functions used to create and manage Ada program libraries, as well as functions used by any program which requires access to information in a program library. These facilities are broken down into three CPCI's: the Program Library Interface Packages (PIF.PLIF), the Program Builder (PIF.PBUILD), and the Program Library Tools (PIF.PLTOOLS).

PLIF defines the physical structure of the program library and provides a complete set of functions which can be used by any program which requires access to such a library, including the compiler, the program builder, and the program library tools.

PBUILD is analogous to the linker in a conventional programming system: it is invoked by a user in order to create an executable program. PBUILD operates in two phases. The first phase, program completeness, guarantees that the program is complete and consistent before the second phase, the linker, assembles an executable program from a collection of compiled units in the program library and binds relocatable symbols to their load-time values.

PLTOOLS is a collection of programs which provide the functions users need in order to manage a program library.

2. Applicable Documents

2.1 Program Definition Documents

Requirements for Ada Programming Support Environments, "STONEMAN," Department of Defense, February 1980.

Revised Statement of Work, 15 March 1980.

Reference Manual for the Ada Programming Language, Draft Standard Document, U.S. Department of Defense, July 1982.

2.2 Inter-Subsystem Specifications

System Specification for Ada Integrated Environment, AIE(1).

Computer Program Development Specifications for Ada Integrated Environment (Type B5):

- Ada Compiler Phases, AIE(1).COMP(1).
- MAPSE Command Processor, AIE(1).MCP(1).
- KAPSE/Database, AIE(1).KAPSE(1).
- MAPSE Generation and Support, AIE(1).MGS(1).
- MAPSE Debugging Facilities, AIE(1).DBUG(1).
- MAPSE Text Editor, AIE(1).TXED(1).
- Virtual Memory Methodology, AIE(1).VMM(2).

Technical Report (Interim) IR-684.

2.3 Military Specifications and Standards

Data Item Description DI-E-30139, USAF, 14 July 1976.

2.4 Miscellaneous Documents

An Incremental Programming Environment, Peter H. Feller and Raul

Medina-Mora, Department of Computer Science, Carnegie-Mellon University, April 1980.

DIANA Reference Manual, G. Goos and Wm. A. Wulf, editors, Institute Fuer Informatik II Universitaet Karlsruhe and Carnegie-Mellon University, March 1981.

3. Requirements

3.1 Introduction

This section provides the set of requirements for the AIE Program Integration Facilities. This includes the performance and interface specifications to which they must comply.

3.1.1 General Description

Program integration occurs in two phases. During compilation, it is the process of analyzing a single compilation unit in the context of a number of compilations which have come before. During program building, it is the process of assembling a number of separately compiled program units into a complete executable program. Program integration occurs within a program library.

The Program Integration Facility allows the compiler to perform the first phase of program integration. While processing a given compilation unit, call it P, the compiler will access the program library in order to obtain information about the separately compiled program units used by P. If the compilation is successful, the compiler will update the library with the new information which resulted from compiling P. This new information will include the abstract syntax tree of P, a DIANA tree, a "compiled unit" (including relocatable object code), as well as some new dependencies between P and program units already in the library. The program integration facility must provide functions which allow the program library to be used in this way.

The Program Builder performs the second phase of program integration, where a complete executable program is constructed from the separately compiled pieces stored in the program library. This requires access to compiled program units stored in the library and the dependency information stored with them.

The Program Library Interface Packages (PIF.PLIF) is provided to address these requirements.

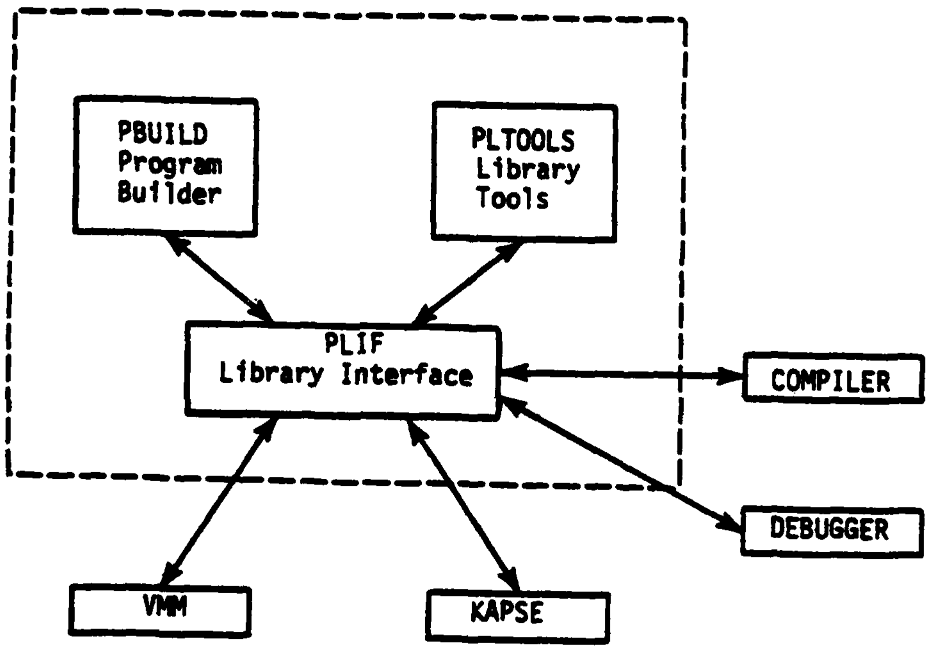
A minimal set of tools must be provided to users for creation and management of program libraries. These tools (PIF.PLTOOLS) provide the mechanism through which users create, copy, and delete program libraries, as well as a means by which the contents of a library may be examined or analyzed.

3.1.2 <u>Peripheral Equipment Identification</u>

Not applicable.

3.1.3 <u>Interface Identification</u>

Figure 3-1 shows the relationships of the PIF to other parts of the AIE. Program interfaces are described in detail in section 3.2.4 below.



32183447-3

Figure 3-1. Program Integration Facility Interfaces

3.2 Functional Description

3.2.1 Equipment Description

Not applicable.

3.2.2 Computer I/O Utilization

Not applicable.

3.2.3 Computer Interface Block Diagram

Not applicable.

3.2.4 Program Interfaces

3.2.4.1 Program Library Structure Interface

The Program Library Structure is designed to support the development and maintainance of very large software systems. The design anticipates the problems inherent in managing a large system in the face of constant modifications and revisions to its components as well as tracking the, perhaps many, total system versions over a long period of time.

3.2.4.1.1 Ada Program Library Representation

Conceptually, an Ada program library is a totally isolated set of interrelated compilation units. However, to allow more sharing of units across libraries, without violating the defined semantics for Ada program libraries, the PIF represents each program library using a "primary catalog" with links to other objects which serve as "resource catalogs" to this primary catalog. Each "catalog" contains an Object Reference Table providing access by unit name to the database objects used to represent the compilation units.

Resource catalogs are sets of units which represent the traditional notion of a "library," like a math library, or an I/O library. It is anticipated that the units of a resource catalog are relatively stable, while the primary catalog is in flux as new source is compiled.

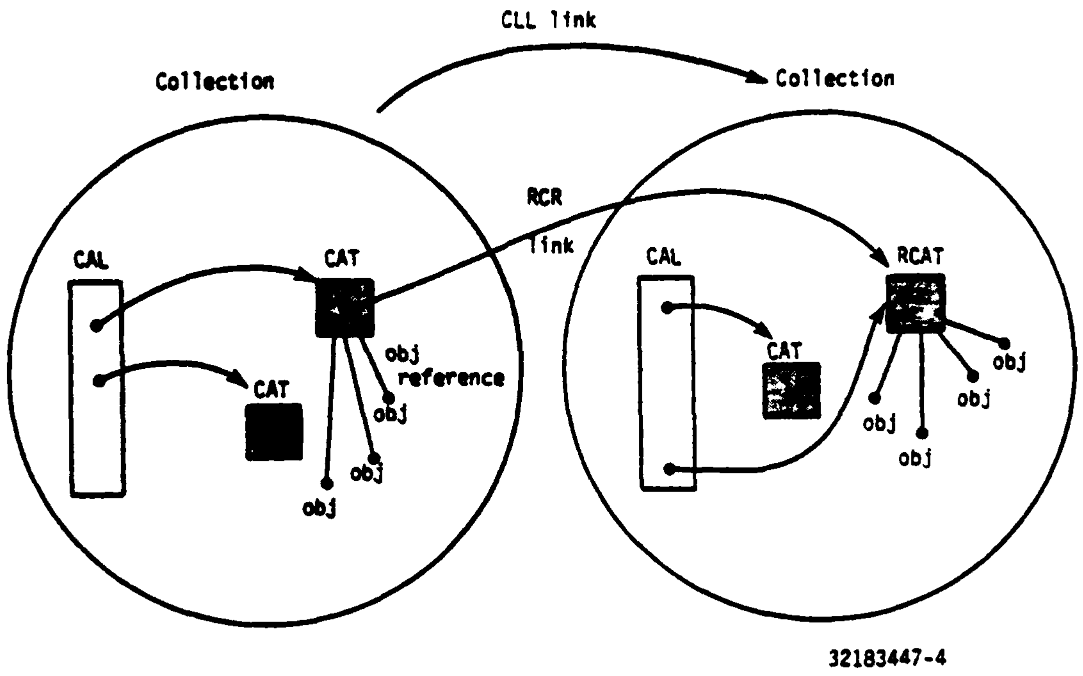
Figure 3-2 provides an overview of the program library structure. Each of the structures in the figure will be discussed in more depth in the following sections.

3.2.4.1.2 Ada Compilation Units -- Ident/Form

In general, an Ada compilation unit can be distinguished by its library unit name, its subunit simple name (if any), and a spec vs. body indicator. This group of information we call the "Ident" of the unit.

For example:

Full Unit Name	Spec vs. Body	Ident
A.B.C.D	Body	A.D
P	Spec	P.IS-SPEC
P	Body	P
P.Q	Body	P.Q



CAL	-	Catalog Access List
CAT	-	Catalog
CLL	-	Collection Link List
Obj	-	Library Object
RCAT	-	Reference Catalog
RCR	-	Resource Catalog Reference

Figure 3-2. Overview of Program Library Structure

A particular unit might also appear in a library in various processing stages or "forms," such as "source," "abstract syntax tree(AST)," "DIANA," "object module," "executable," "documentation," etc. The combination of the Ident and the Form is sufficient to uniquely identify a current member of a library.

3.2.4.1.3 Initial Form and Consistency

For every unit (specified by some Ident), there is required to be some form which is independent of all other objects within the library. For this Ident, this is considered its "initial" form. In general, this will be the AST for Ada compilation units.

Library consistency is defined in terms of initial forms. Every object maintains a complete list of all initial forms from which it was built (The "initial form precursor list"). If any of these initial forms have been replaced in the catalog, the derived form is considered out-of-date.

The advantage of relying on initial forms is that intermediate forms (such as object modules) may be deleted from the program library (to conserve space, presumably), without affecting the "up-to-date-ness" of the final, most useful form (such as an executable).

3.2.4.1.4 Primary and Resource Catalogs

As mentioned above, rather than referencing all the compilation units of an Ada program library from a single directory-like database object, references to the units are spread among a primary catalog, and a set of relatively stable resource catalogs.

Because of the heavy inter-dependence of units within an Ada program library, it is not practical to do version/revision selection on a unit-by-unit basis. Instead, the PIF library structure supports versions and revisions of catalogs. Two versions or revisions of a catalog may refer to many of the same units, with only small differences in the source submitted to the two. However, the DIANA and Object Modules produced from even unchanged source will generally be different if some depended-on unit has been changed. Thus, it makes more sense to treat a revision on a catalog basis, even if the change has been restricted to the sources of a small set of units.

3.2.4.1.4.1 Unit-name Prefix Set (of a Resource Catalog)

To limit the number of catalogs in which a unit might appear, each resource catalog defines a small set of unit-name prefixes, such as "IO_", "INPUT_", "OUTPUT_" . All units associated with the resource catalog are required to have Idents which conform to the catalog's prefix set, such as "IO_EXCEPTIONS", "INPUT_OPERATIONS.SUB_PACKAGE", "INPUT_DEFS", "OUTPUT_FORMATTING.BODY", etc.

3.2.4.1.4.2 Interface and Implementation Catalogs

A further distinction is made between "interface" catalogs containing units needed for compile-time reference (specs and in-line bodies), and "implementation" catalogs containing units needed only at program build time (non-in-line bodies, and specs and bodies of "implementation-specific" units). A resource implementation catalog must specify its interface catalog, and must restrict its own units to the same unit-name prefix-set as its interface, to avoid colliding with names used in other catalogs.

To provide implementation independence at compile time, resource interface catalogs may only be linked to other resource interface catalogs (to avoid "indirect" implementation dependencies). Resource implementation catalogs may also only be linked to resource interface catalogs. Only a primary catalog may be linked to resource implementation catalogs (as well as to any interface catalogs).

The benefit of these restrictions is that, as long as a resource interface catalog remains stable, multiple implementations may exist for it (both over time, and simply as different co-existent versions), without repeatedly making obsolete compiled units within user's libraries. Only executable load modules become dependent on the particular resource implementation catalog chosen, meaning that only a re-Build rather than a complete recompilation is needed to track updates to the resource implementation.

3.2.4.1.5 Object Identification and Catalog Selection

A primary catalog may only link to resource interface catalogs with non-overlapping unit-name prefix-sets. Also, for each resource interface catalog, the primary catalog may link to one implementation of it.

Hence, given the Ident/Form for a unit, at most two catalogs, accessible from the primary catalog of a library, could hold it. If the Ident falls within the prefix set of a particular resource interface catalog, then either that catalog, or the implementation of it would have to contain the unit. If the Ident falls within the prefix set (if present) of the primary catalog then either that catalog or its interface (if any) would have to have the unit. Note that if the primary catalog does not specify a prefix set, it contains (by definition) anything not contained in the prefix sets of the resource catalogs to which it is linked.

If both an interface and implementation catalog might contain a unit, the interface catalog is always searched first. When a unit is found in a catalog other than the primary catalog, the unit is considered "read-only," and may not be replaced or updated. Any new submission to the compiler, or new forms produced as a side effect of compiling or linking, must all go into the primary catalog, and must abide by the unit-naming restrictions implied by the prefix sets.

3.2.4.1.6 Collections

To provide a convenient unit for resource allocation, access control, and revision maintenance, the available database storage may be organized into "collections." A collection is simply a set of database objects, each object uniquely identifed by a never-reassigned key, called the ObjID, that represents a particular revision of a particular object within the collection.

Catalogs are themselves represented as objects within a collection, and a restriction is made that all units referred to directly by a catalog's Object Reference Table also be objects within the same collection. Thus the only way to make cross-collection references is via a cross-collection catalog link.

To aid in catalog management, all catalogs of a collection to be available as resource catalogs must be entered ("promoted") into a master "Catalog Access List" (CAL) associated with the collection. The catalog's entry in the CAL defines the name/version/revision of the catalog.

For a particular name and version, all revisions of the catalog are limited to the same unit-name prefix set. The latest revision is the one normally used, although it is possible to specify an older revision explicitly. When a resource catalog is to be updated, a private catalog is "derived" from the old revision, corrected or enhanced as appropriate, and then "promoted" as the new revision.

Because many of the objects in the PIF library structure are implemented using VMM primitives, the collection is a convenient level at which to assign VMM segment numbers, which must be unique across all VMM subdomains simultaneously addressable within the same domain (see AIE(1).VMM(2)).

3.2.4.1.7 Detailed Structure

3.2.4.1.7.1 Catalog Access List (of a Collection)

A Catalog Access List (CAL) provides linkages to all resource catalogs available in a collection. For each catalog which has been promoted, an entry exists in the CAL containing:

1. the catalog name;

2. its list of prefixes;

3. a list of revisions for that name (including revision number and ObjId);

4. an indication of whether it is an interface or implementation catalog;

5. either the name of the default inplementation catalog (if interface) or the name of the related interface catalog (if implementation).

3.2.4.1.7.2 Collection Link List (of a Collection)

Also associated with each collection is a Collection Link List (CLL) which is a list of all collections that are available for reference from catalogs within this collection. This includes the identifications of these collections and a list of the ObjIds of the catalogs that make references.

Each collection also has an associated list of rules which define how a particular form of a library object is derived from other forms of the same object. Each rule identifies the precursor form, the target form, and the tool which is used to transform it (e.g. AST -> FE_DIANA:Compiler_ Sem_Phase). These rules are referenced by the bring_up_to_date function invoked by tools which require the existence of certain forms (e.g. the Linker requires one or more object modules).

Each collection also maintains information regarding the VMM segment numbers which have been assigned to it. This includes a list of free segment ranges.

3.2.4.1.7.3 Object Reference Table (of a Catalog)

Each catalog includes an Object Reference Table, which can be viewed as an ident/form matrix referencing program library units (Idents) in various processing stages (Forms).

Figure 3-3 is an example of an Object Reference Table viewed as a matrix in which the rows are the Idents of the units, and the columns are the Forms of the units:

		<--- Forms --->				
		Initial	. . .	DIANA	. . .	Executable
I D E N T S	ident 1	obj 11	. . .	obj 1i	. . .	obj 1j
	ident 2	obj 21				
	ident 3	obj 31	. . .	obj 3i		

Figure 3-3. Object Reference Table

3.2.4.1.7.4 Resource Catalog Reference List (of a Catalog)

Also associated with each catalog is a Resource Catalog Reference (RCR) list which is used to provide linkages to resource catalogs. This list also contains "indirect links" to resource interface catalogs. "Indirect links" are recorded in a primary catalog to all of the resource (interface) catalogs to which any directly linked resource catalog itself is linked.

The RCR contains for each catalog

1. collection identification,
2. catalog ObjId,
3. catalog name/version,
4. revision number,
5. indication of whether tied to specific revision, or wish to track new revisions on "update" (see below),
6. list of prefixes,
7. indication of whether implementation or interface,
8. indication of whether directly or indirectly linked (only units of directly linked catalogs are available to Ada WITH clauses).

To provide efficient access, based on the prefix of the unit name, a secondary index to the RCR is provided (the "prefix index"), keyed off of the prefixes, providing direct access to the RCR entries for the interface and implementation catalogs associated with a particular prefix. Figure 3-4 is a user's view of an RCR list with prefix index.

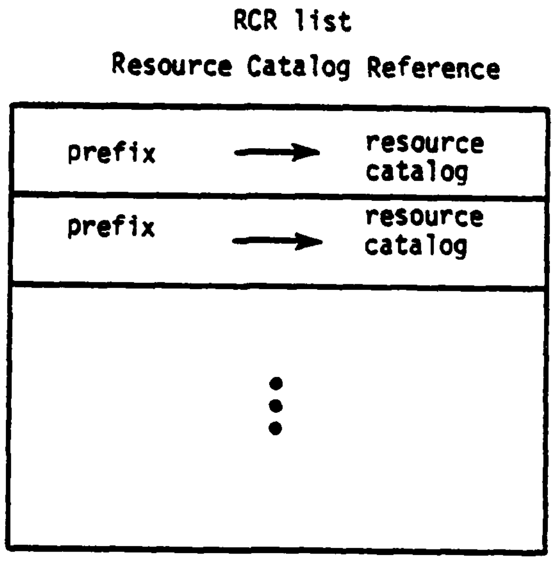


Figure 3-4. RCR List with Prefix Index

3.2.4.1.7.5 <u>RCR Update and Catalog Derivation/Promotion</u>

Periodically, a user may wish to "update" the links in a catalog's RCR. At this time, an entirely new RCR (and prefix index) is built, attempting to re-link to the latest revision of each resource catalog (unless specifically tied to an older revision). If this can be done consistently (i.e. no indirect links to interface catalogs refer to a different revision than the new direct link), then the new RCR replaces the old. At this point, many of the units within the primary catalog may have become out-of-date with respect to the updated resource catalogs.

To make a catalog accessible to other catalogs as a resource, it must be <u>promoted</u> into the CAL of its associated collection, under a particular name/version. At this time it is specified to be either an interface or an implementation catalog and is assigned a revision number. A new revision of a catalog is created by <u>deriving</u> a new catalog from a previously promoted one, modifying some of its units and promoting this new catalog.

3.2.4.2 KAPSE Interface

The Program Integration Facility uses the KAPSE database facility in order to represent the objects and their attributes in the program library and, like any Ada program, the programs that make up the PIF require use of the KAPSE for program invocation and run-time support (KAPSE.MULTPROG and KAPSE.RTS). These are the only functions required from the KAPSE.

In addition, the Linker (PIF.PBUILD.E) is required to conform to the load module format specified by the KAPSE.

3.2.4.3 VMM Interface

The PIF stores information in the Program Library in data structures defined using Virtual Memory Methodology (AIE.VMM). This enhances portability (rehosting) of the PIF, and provides convenient access to these structures.

The basic VMM addressing elements are:

Domain	The largest unit of data in VMM, defined by a collection of Subdomains for the duration of program execution.
Subdomain	Defined before and after program execution by a KAPSE database resident external page file. Each subdomain consists of some number of data segments.
Segment	A logistic device to facilitate flexible VMM Locator addressing.
Page	Each segment is composed of 32 or fewer pages, of 2048 bytes each. Software paging is done at this level.
Locator	A locator is a pointer into the VMM data structure. It consists of a segment number, a page number, and page offset.

Program integration occurs completely within a single VMM domain, where each Ada program unit in the library is assigned a segment number. The number of segments available in a VMM domain is limited, making it necessary to reuse segments when the information contained in them becomes obsolete. For this reason, a map of segment numbers currently in use is kept in each "collection" managed by the PIF Library Configuration Management CPC.

3.2.4.4 Object Module Format Interface

This section defines the format of the object module associated with each compilation unit. An object module defines machine instructions and initial values for data needed to implement a particular Ada compilation unit, as well as definitions and references to entities accessible across compilation units, whose addresses are typically unknown until link-time.

An object module is implemented as a single VMM subdomain, with each construct within it implemented in terms of VMM virtual records, called nodes, as described in the following sections.

3.2.4.4.1 Control Sections

The construct of primary concern both to the code generator and the linker is the "control section" (CSECT). A CSECT is a block of storage units that are to be allocated contiguous space in memory by the linker. The code generator defines the size and external name of each CSECT, and also defines any or all of the storage units within. The storage unit is as defined in package SYSTEM for the target machine. For example, on both the 4341 and 8/32 target machines, STORAGE_UNIT:=8 defines the storage unit as 8 bits. The term "SU" will be used in the following sections to mean:

```
Type SU is 0..2**STORAGE_UNIT-1;     -- defines a byte for
                                        4341, 8/32

For SU'SIZE use STORAGE_UNIT;        -- length specification
```

The node defining a CSECT is uniquely identified by the following attributes:

1. the VMM locator for the associated DIANA construct (will refer to the spec rather than the body, because only that is known to its users); and

2. a CSECT "selector" (spec elaboration, body elaboration, body call, etc.), to distinguish among the multiple CSECTs associated with a single DIANA construct.

These two attributes effectively represent the external "name" of the CSECT.

The node defining a CSECT has the following additional attributes:

1. A Global vs. Local flag -- this determines whether this CSECT definition is directly accessible outside of this object module;

2. a list of compile-time SU definition (CSUD) nodes;

3. a list of link-time SU definition (LSUD) nodes;

4. the size of the CSECT (in SUs);

5. the default fill-value for the CSECT (for otherwise uninitialized SUs);

6. an alignment (in SUs) -- the linker will place the CSECT at an address divisible by this number;

7. a pure vs. impure indicator (pure CSECTs are subject to memory protection and sharing at load time). For pure CSECTs, an additional attribute indicates whether the CSECT is data-only, instructions-only, or a combination (affects memory descriptor set-up on some machine architectures);

8. an optional address specification -- the linker will place the CSECT at the designated address;

9. an optional group CSECT specification -- the linker will place this CSECT and all others with the same group CSECT contiguous in storage after the group CSECT (this allows a primitive grouping of CSECTs). The length (see LLEN below) of the group CSECT is considered to be the sum of the directly-defined size of the group CSECT plus the sum of all of the lengths of the CSECTs specifying it as their group CSECT (this is intentionally a recursive definition);

10. an optional overlay root CSECT specification -- the linker will place this CSECT and all others with the same overlay root CSECT at the same address, immediately after the root CSECT, effectively overlaying one another. The length (see LLEN below) of the root CSECT is considered to be the sum of the directly-defined size of the root, plus the maximum of the lengths of the CSECTs specifying it as their overlay

root CSECT (this is intentionally a recursive definition);

The PIF does not support the definition of storage units for a single CSECT from more than one object module. Instead group and overlay root CSECT specifications are used to control cross-object-module CSECT contiguity.

3.2.4.4.2 ENTRY definitions

An ENTRY definition node is used to define other link-time values, usually as an offset with respect to some other CSECT or ENTRY. ENTRYs are used in a situation where reference to a DIANA construct may occur from another compilation unit, but the reference is to an unknown CSECT or unknown offset within a CSECT. The referencing unit does not know the displacement within the CSECT, so the EXTERN reference (see below) to an ENTRY node serves as a place holder. Each ENTRY is associated with some DIANA construct.

An ENTRY definition node is uniquely identified by the following attributes:

1. the VMM locator of the associated DIANA construct;

2. an ENTRY selector (analogous to a CSECT selector), to distinguish the ENTRY from other ENTRYs or CSECTs associated with the same DIANA construct.

These two attributes represent the external "name" of the ENTRY.

The following additional attributes are associated with each entry:

1. Global vs. Local flag -- This determines whether this ENTRY is accessible outside of this object module; and

2. the link-time value of the ENTRY node, defined in general in terms of other ENTRYs or CSECTs, as a single "link-time expression node" (LTEN -- see below).

3.2.4.4.3 EXTERN Reference node

An EXTERN reference node represents the value of some externally-defined CSECT or ENTRY. It is not required to distinguish between EXTERNs referring to CSECT addresses and

EXTERNs referring to ENTRY values.

An EXTERN reference node consists of the "name" of the externally-defined CSECT/ENTRY, which includes a DIANA VMM locator, and a CSECT/ENTRY selector.

3.2.4.4.4 External Definitions and References

Taken together, the set of CSECT definition nodes, ENTRY definition nodes, and EXTERN reference nodes represent what is conventionally the "symbol table" of the object module.

A CSECT or ENTRY definition is always associated with some DIANA construct, whose VMM locator forms part of its "name." The rules of Ada guarantee that the CSECT or ENTRY is defined either within the object module associated with the unit in which the DIANA construct appeared, or within an object module associated with some secondary unit of the same library unit. It is not necessary to search the entire program library to determine where a CSECT/ENTRY will be defined. It is determined fully by the DIANA VMM locator, the selector, and the set of secondary units associated with the unit in which the DIANA construct appeared.

3.2.4.4.5 Storage Unit Definitions

A CSUD (compile-time SU definition) node is used to define the compile-time values of a contiguous array of storage units within the CSECT to which it is attached. (This corresponds to the .TXT record of a 4341 object module.) It has the following components:

1. a displacement (in SUs) within the CSECT; and

2. an array of SUs containing the machine instructions and/or data.

The compiler may define the storage units of a CSECT with one or more CSUD nodes. Some or all of the storage units of a CSECT may be left undefined, in which case a specifiable fill-value will be used.

An LSUD (link-time SU definition) node is used to define the link-time values of storage units in the CSECT to which it is attached. (This corresponds to the .RLD record of a 4341 object module, which defines relocation data.) The value is specified with an arbitrary expression containing a limited set of operators and operands as described below. An LSUD node contains the following attributes:

1. A displacement (in SUs) within the CSECT where the link time value should be deposited;

2. A size (in SUs) of the link time value (right adjusted in the field);

3. A locator of a link-time expression node (LTEN) defining how the linker is to compute the value.

The link-time value is typically 1 to 4 storage units in size, and typically contains an address. However, the linker will also assign values representing Ada exception identities, stack frame sizes, and other entities which the compiler cannot reasonably compute (due to separate compilation).

3.2.4.4.6 Link-Time Expression Nodes (LTENs)

The link-time expression nodes, LOPR, LREF, LLIT, LLEN, are used to represent the definition or computation of a link-time value. The value is referenced by an LSUD node or an ENTRY definition. A value of arbitrary complexity may be represented by a tree structure with LOPR branch nodes and LREF, LLIT, and LLEN leaf nodes. A simple value may be represented with a single leaf node. Thus complex address constants or other link-time values may be defined by the compiler and computed by the linker.

3.2.4.4.6.1 LOPR node

An LOPR node represents an operation. It has the following components:

1. a binary operator (plus,minus,times,divide);

2. a VMM locator for the left operand LTEN;

3. a VMM locator for the right operand LTEN.

3.2.4.4.6.2 LREF node

An LREF node represents the value of a link-time symbol, consisting of a VMM locator to the CSECT, ENTRY, or EXTERN node within the object module "symbol table."

3.2.4.4.6.3 LLIT node

An LLIT node represents a literal. It contains the literal value as an implementation-defined integer type as its only attribute.

3.2.4.4.6.4 LLEN node

An 'LEN node represents the "length" of a CSECT. Its value is, in general, the same as the "size" of the CSECT as specified in the CSECT definition. However for CSECTs referred to as a group CSECT or as an overlay root CSECT, the length is extended to include other CSECTs made contiguous with the CSECT.

An LLEN node consists of a VMM locator to a CSECT or EXTERN node within the object module symbol table. It is a link-time error if the EXTERN node actually refers to an ENTRY rather than a CSECT. (i.e. its VMM DIANA pointer, and its selector).

3.2.4.4.6.5 Use of link-time expressions

Traditional linkers support only limited operations on addresses (such as the addition of a signed constant). By providing for more general link-time expressions, the PIF object module format allows for the creation at link-time of more complicated values, such as a byte pointer to the odd byte of a word, on an otherwise word-addressed machine. The word address for this hypothetical example is represented by an LREF node; an LOPR node (times) points to the LREF node and to an LLIT node (value 2), instructing the linker to double the word address; another LOPR node (plus) points to the LOPR (times) node and to an LLIT node (value 1 for the odd byte).

This generality ensures that the linker will be suitable for handling code generated for various target machine architectures.

3.2.4.4.7 Overall Object Module Organization

The root node for each object module contains a list of all CSECT definition nodes, a list of all ENTRY definition nodes, and a list of all EXTERN reference nodes. From the list of LSUDs associated with each CSECT, the linker can determine all cross-CSECT references (LREFs and LLENs).

3.2.4.4.8 Run Time Routines

The Ada compiler generates code that calls run time routines for certain language constructs (allocators, tasking, exception raising, etc). These routines must be available to the linker, and there must be a convention for the compiler to use for the run time routine CSECT references. It is desirable to code as much of these routines in Ada as possible, yet not allow direct reference to them by Ada source code. (For example, if an allocator "NEW t" were translated by the code generator into a call on a function "alloc_storage(t'size)" defined in the run time library, the Ada specification must be defined in such a way as to prevent a programmer from calling "alloc_storage" directly.)

This problem is resolved by placing the specifications of the run time library in the private part of a predefined package specification (for example package SYSTEM). Thus, only the compiler itself will have visibility to the run time routines. This solution adds no additional complexity to the linker, since the reference mechanisms are the same as to user-compiled Ada programs.

3.2.4.5 Compiler Interface

The proper context for the current compilation unit is established by the compiler through calls to the program library interface packages, which will perform the analysis to determine if the needed library objects are up-to-date with respect to the most recent submittals to the program library. If the needed library objects are not in a consistent state, the program library manager can optionally invoke the necessary compiler phases to bring the needed library objects up to date.

As a result of the lexical/syntactic phase of the Front End (COMP.FE) of the compiler, the abstract syntax tree (AST) of a compilation unit is entered into the program library. Compilation can be suspended at this time, and the compiler can be called subsequently to complete the compilation from this AST.

The source submitted for compilation may consist of several compilation units. The LexSyn phase splits a submittal into individual compilation units.

The Sem phase of COMP.FE performs semantic analysis which results in a DIANA tree, which is further attributed by the Middle Part (COMP.MID), and entered in the program library. The DIANA representation of a unit in the program library may become outdated if a unit it is built from is re-submitted.

Both COMP.FE and COMP.MID refer to other compilation units during the creation of a library object, and thereby create dependencies which must be recorded in the program library.

The PIF allows multiple instances of the compiler to process program units in a single program library at one time. A primitive locking mechanism is provided automatically to insure that concurrent processing can be supported.

For each compilation unit processed, the Back End (COMP.BE) produces an "object module," which includes both relocatable object code, organized as several "control sections," and a symbol table for externally defined or referenced addresses and values. In general, a single compilation unit may contain a number of nested Ada program units.

In order to allow for sharing of late-supplied generic body templates, units with generic instantiations have an additional object module created within the program library, defining entries from the generic instantiation in terms of entries into a specific implementation of the generic body.

Units with generic templates also have an additional set of object modules, one for each implementation of the template.

B5-AIE(1).PIF(1)

The phases of the compiler will be entered as tools in the rules associated with a collection used for Ada program libraries, and must conform to the parameter interface associated with the Rules Interface.

3.2.4.6 DBUG Interface

DBUG requires the following information:

1. Relocation Map: records the placement of each compiled unit in the program image;

2. The compiler generated statement table.

The Relocation Map gives the base address of each compiled unit in the executing program. This map is created by the Linker. DBUG uses this to determine the new scope when the user gives a SCOPE ENCLOSING or SCOPE TO command.

A statement number table is created by the compiler, and stored as a CSECT within the object module. DBUG uses this table, in conjunction with the Relocation Map produced by the Linker, when directing the run time system to activate or deactivate specific breakpoints.

3.2.4.7 Unit Lister Output Format

The Unit Lister Output Format is the standard interface for human-readable compiler and cross-reference listings. Unit Lister Output may include source text, symbol table attributes, a cross reference, and an assembly listing.

3.2.4.7.1 Source Text Listing

Each line of the source text listing includes the following fields:

1. The counts of the nesting depth. There are two counts, the nesting depth of the current program unit and the nesting depth of the current statement.

2. The statement number. Numbering starts from the beginning of each subprogram, task, generic unit, and package in the compilation unit. Both statements and declarative items are numbered.

3. The source program text.

4. The current scope. This field contains the current program unit name, truncated if necessary.

5. A cross reference for identifiers in the source text. Each occurrence of an identifier in a line of source text has a corresponding cross reference entry in this field. If the identifier is being declared on this line, the cross reference entry is an "=" character. If the identifier was declared previously within the compilation unit, the cross reference entry is the listing page number at which it was declared. If it is declared later in the compilation unit, the entry is a "?" character. Otherwise, the entry is a letter which corresponds to a WITHed library unit. This field is continued on succeeding lines if necessary. A legend of the withed library unit letters and names is supplied in the listing.

Messages generated from processing errors are formatted within the source lines. If the precise location of the error is known, a line containing the single character "^" is printed under the line producing the error, with the pointer beneath the leading error term. The error message is then printed, with the severity and the statement number incorporated. A line is also included which indicates the page and statement number of the previous error.

The listing includes a final section which gives an error summary and a tally of compilation statistics.

3.2.4.7.2 Symbol Table Attributes

The symbol table attributes listing is an alphabetic list of all the identifiers explicitly or implicitly declared in the unit. Each identifier is listed with the statement number of its declaration, the name of its immediate scope, its type information and other information specific to the identifier.

If the type is not an identifier, the type information will be a description such as Type, Task, Function, Block name, Label name or Exception. Types declared in other library units will have a flag. Other information may be supplied depending upon the identifier. The base type is given for subtype and derived type entries. Mention is made if the identfier is a constant or declared a number, has a representation clause, is a private type or a generic instance, etc.

3.2.4.7.3 Cross Reference

The cross reference is similar in format to the symbol table. Its header includes the same fields as the symbol table: identifier, statement number of the identifier's declaration, immediate scope, type and additional information. If the type or immediate scope is declared in another compilation unit, the name is flagged as it was in the symbol table.

Following the header, the cross references are listed. The name of each program unit which contains references to the identifier is followed by the statement numbers of the references. Statement numbers at which the identifier is written to or appears as an out parameter are flagged with an asterisk.

3.2.4.7.4 Assembly Listing

The assembly listing is similar to the listing produced by the 4341 assembler. It contains relative location and object code fields to the left of basic assembly language mnemonics and operands. If the source text listing is also generated, the assembly and source text listings will be intermixed.

3.2.5 Function Description

The Program Library Interface Packages CPCI consists of the following components:

1. Library Object Identification: This CPC provides the means by which a representation of a compilation unit (eg. Abstract syntax tree, DIANA tree, or Object Module) is identified and retrieved from a program library.

2. Library Object Dependency Manager: This component tracks all dependencies between objects in the program library.

3. Library Object Allocation Package: This component is used to allocate and reclaim VMM segment numbers for library components.

4. Library Configuration Management: This component provides primitives for the creation, manipulation, and deletion of catalogs and collections.

The Program Builder CPCI consists of the following components:

1. Builder: Calls the Program Completeness Checker, Body Generator, the Preamble Generator, and the Linker as necessary in order to create an executable program.

2. Preamble Generator: Builds a subprogram body which interfaces the main subprogam unit to the KAPSE.

3. Program Completeness Checker: Checks that all bodies required by specs or stubs accessible from a main program unit, have been provided. A list of the names of those bodies that are missing is produced as its output. This list is used to drive Body Generation, as part of the first phase of program building.

4. Body Generator: Creates a null body when a spec or stub exists but a body does not. This subprogram will be invoked by the Builder, as well as recursively to generate null bodies for components of missing package bodies.

5. Linker: Combines object modules into a single load module, and evaluates link-time addresses, to within a single relocation constant for the entire pure and impure parts of the program.

The third CPCI making up the Program Integration Facility consists of the following Program Library Tools:

1. Program Library Manager: This program provides access for the interactive user to the various primitives of the Program Library Interface Packages. It allows the user to create, manipulate, display, and delete the various objects forming a program library.

2. Change Analyzer: This program computes the differences between different versions of the same program unit, by walking the two DIANA representations, and creating a VMM locator mapping set.

3. Recompilation Minimizer (MAP): This program uses the Change Analyzer and the Dependency Manager to mark DIANA and object modules as up-to-date, without re-compilation, if sufficiently minor changes have occurred in units they depend on.

4. Link Map/X-Ref Lister: Produces a human readable link map, including linker error messages (if any) and global cross reference information.

5. Source Reconstructor: Reconstructs source listings from the DIANA or AST representation of a program.

6. Unit Lister: Produces source, symbol table, and cross-reference listings from information in the DIANA for a compiled unit. This program is invoked from the compiler if an immediate listing is requested, and may also be invoked later by the user for new or additional listings without requiring re-compilation.

7. Foreign Object Module Importer: Installs object modules from systems other than the AIE into the program library.

3.3 Detailed Functional Requirements

3.3.1 Program Library Interface Packages (PLIF)

The PLIF consists of four components: Library Object Identification, Library Object Dependency Manager, Library Object Allocation Package, and Library Configuration Management, and provides all of the primitives for access to the library data structures, as a set of Ada packages.

3.3.1.1 Library Object Identification

This CPC is provided for users of the program integration facility: to access, create, save or delete an object in the catalog. Library object identification allows the user to specify which object to reference or utilize from his catalog or linked catalogs. Information about objects is stored either in a catalog or with the object, depending upon the uses of the information.

3.3.1.1.1 Objects And The Catalog

The catalog refers to compilation unit objects which can be distinguished by their identification (name of subunit/libunit/is_spec) and by their form (AST, DIANA trees, ...). An object is referenced by using its ident/form. If an object is from another catalog linked to this catalog (determined from the prefix index associated in the resource catalog reference, RCR list), the specification of a collection and catalog is needed in addition to the ident/form.

Besides containing a table matching ident/forms to their corresponding objects, the catalog also contains information needed to identify itself and the catalogs it refers to. The collection also provides for each object a unique object id, ObjId, to allow locating and referencing objects, even if no longer up-to-date, and to determine whether a given object is the same as the one stored under ident/form. Revisions of an ident/form in a catalog will have different ObjIds. The object table in the catalog has one, if it exists, up-to-date ObjId per ident/form. Every object that is created in the collection has a distinct ObjId and the ObjId is never reused in the collection. The collection and ObjId uniquely identifies any object under the program integration facility. The user has no access to the unique object id.

The catalog's object table contains the list of the most current ident/forms along with their ObjIds. In addition, each object in the catalog has a backup form , which is used in consistency checking. According to the Ada LRM, any compilation/recompilation that detects errors must have no effect on the program library. In an AIE library , whenever a unit is successfully compiled through all phases of the compiler and all objects it depends on have also been successfully compiled, the currently successful object is entered both as the current state of that object, but also as the backup of it as well. If compilation is unsuccessful, then only the current state of the object is updated and the backup left untouched.

The collection also maintains a state variable for each object in the collection, to distinguish objects in the process of being created from those that have completed. Thus if, for example, the compiler front end might ask the library to create a

DIANA object by copying the AST and then allowing the compiler to work on the copy. While the compiler is working on the AST copy, it will be marked in the collection as "in process", and only when it is complete will it be fully accepted into the collection.

Consistency checking uses the backup state information on an object, thus restricting it to successful compilations in conformance with Ada rules. All other references to an object in a catalog use the current state information. This enables the lister to find the most recent listing information.

3.3.1.1.2 Object Information

An object is accessed via a handle which is returned by the routine to open an object. A handle is more than just the ObjId, because the ObjId may be contained in more than one catalog. This would be true, for example, if a catalog was copied from another catalog. The specific content of a handle is implementation dependant and not described here. Whenever the term object is used in conjunction with a routine either as an input parameter or as an output result, what is actually being referred to is this handle.

An object has a variety of information associated with it. It has three precursor lists, one for its direct precursors, one for VMM references and one for deciding whether the object is up-to-date. It also has user parameter information (such as whether optimized code is to be produced), and the VMM information about segment numbers. Figure 3-5 is a schematic view of a library object.

3.3.1.1.3 Library Object Identification Functions

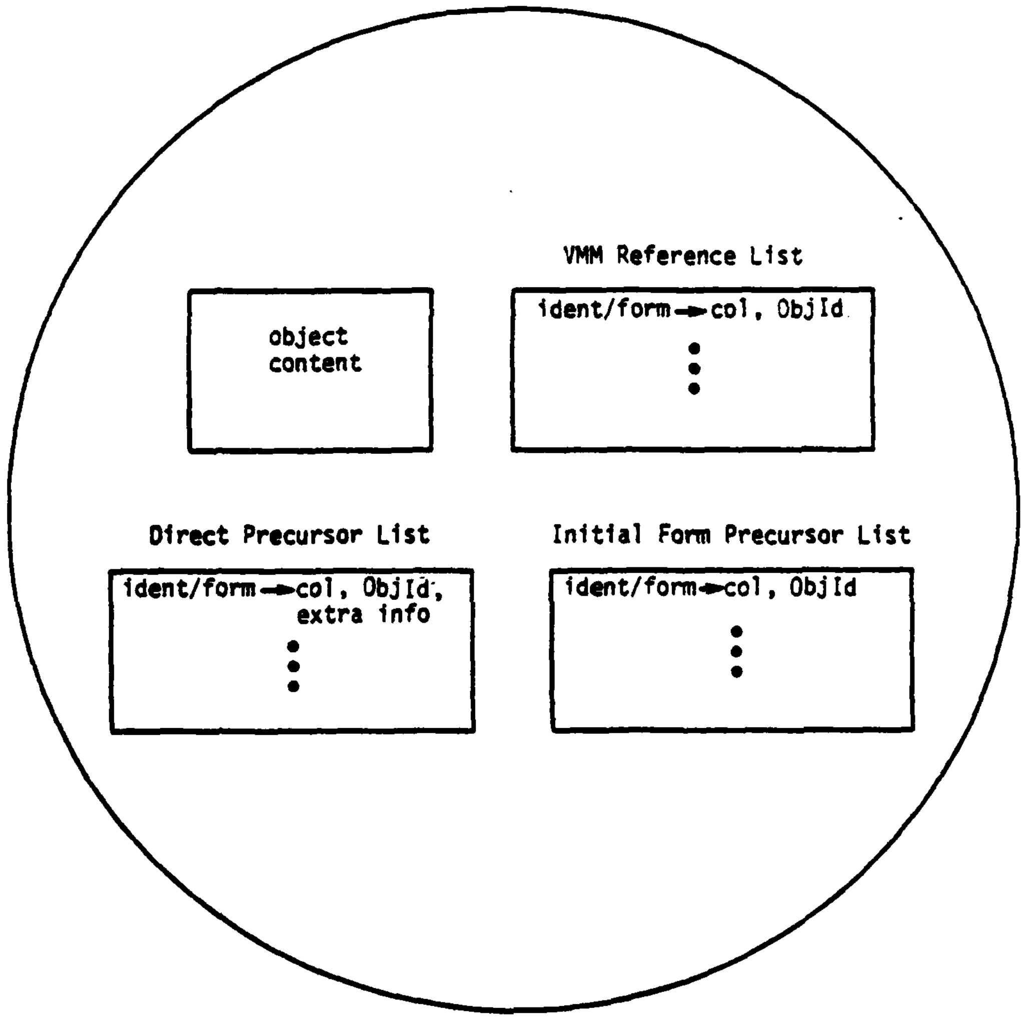
The main functions involved in the library object identifications are addition into, accessing of and upkeep of a library object.

3.3.1.1.3.1 Object Addition

An object may be added into a catalog either by submitting its initial form, creating an "empty" object to be filled in, or copying an object already in the catalog. The object is added using the tool specified in its collection's rules.

3.3.1.1.3.1.1 Inputs

In order to add an object into a catalog, the catalog and the target ident/form in the catalog must be specified. For objects that are copied into a catalog, the object to be copied from must also be specified. If the object is "submitted" into



32183447-1

Figure 3-5. Library Object

the catalog, the form becomes the initial form, a form that does not depend on any other ident/form in the catalog.

3.3.1.1.3.1.2 Processing

Any ident/form that is added into the catalog must conform to the rules of the catalog's collection. When an initial form is submitted into a catalog, the tool used to add the object must conform to the tool specified in the collection's rules for initial forms. If conformance is established, then the ident/form's ObjId is added to the object table of the catalog.

Normally, when an ObjId is created in a catalog, its precursor lists are initialized to null. However, when an object is copied into a catalog, its initial form list, used to help decide whether the object is up-to-date, is also copied from original object's initial form list. Other information relating to the new object is filled in if known depending on whether the object was created by submittal, copied or just simply created.

3.3.1.1.3.1.3 Outputs

A new object is created with its new ObjId after the creation of the new ident/form. The new ObjId is added into the catalog's object table. Initial and default information about the object is attached to the object depending on how it was added.

3.3.1.1.3.2 Accessing Objects

Objects can be accessed in the catalog by opening the object, which returns an object handle (a way to refer to the object). If an ident/form exists in the catalog, but the corresponding object does not, the ident/form can be explicitly brought up-to-date. When an object is opened, all the information associated with the object is made accessible.

If an ident/form does not exist directly in the catalog, there are two ways to access an object in other catalogs. One way is by using the direct link in the RCR of the main catalog and the other is by using the indirect links to resource catalogs referenced by other direct resource catalogs.

Objects once opened can be closed.

3.3.1.1.3.2.1 Inputs

To open an object, the catalog and its ident/form are needed. To close an object, the object handle is provided. If the ident references a resource catalog, how the ident is to be reached, directly or indirectly, must also be specified.

3.3.1.1.3.2.2 Processing

For an object to be opened, its object handle must be found. If the compilation unit name of an object begins with a prefix reserved to a particular (direct or indirect) resource catalog, then that catalog referenced in the RCR, resource catalog reference list, is the one to consult. Otherwise, the current catalog's object reference table is the place to start. In the resource catalog, the resource interface catalog is referenced first and the resource implementation catalog second. If the resource catalogs are to be reached directly, then if the object is not found, the object can not be opened. If the resource catalog is to be opened indirectly, the catalogs referenced by the resource catalogs are also searched if need be.

An object is closed when no more references to that object are necessary; the object handle to the object is released.

3.3.1.1.3.2.3 Outputs

If the ident/form is opened, the handle to the object is returned. If the ident/form is not found in the catalogs specified, the appropriate message is returned. If the object does not exist in that catalog, the catalog may be brought up-to-date; and if the object was brought up-to-date successfully the object can be opened. The object can be closed after all references to it are complete.

3.3.1.1.3.3 Deleting And Saving Catalog Objects

A routine is provided to delete or save a catalog object. Deleting is desirable if an object becomes inaccessible because its ObjId is not known to anyone or if an old ObjId is to be withdrawn from the user's view. Deleting is a permanent erasure of the object. Saving, (because of archiving, saving space, etc.) involves moving the actual object contents somewhere else, thus making them recoverable.

3.3.1.1.3.3.1 Inputs

The object must be located, its ObjId known before it can be removed.

3.3.1.1.3.3.2 Processing

If the ObjId is to be deleted, it is completely removed. When other objects or catalogs reference this object they will not find it. When the object is to be saved, a stub is left such that it can be retrieved again.

3.3.1.1.3.3.3 Outputs

If the ObjId is to be deleted, the object referenced by the ObjId is not accessible to anyone. If the object is saved, it can retrieved at later date.

3.3.1.2 Library Object Dependency Manager

This CPC is the set of routines necessary to keep the library in a consistent state, as defined by the Ada LRM. Precursor lists attached to catalog objects and rules attached to the catalog's collection are essential in maintaining the library in a consistent state. Precursor lists are used to check whether an object is up-to-date. Rules are used to create/recreate objects that are needed in order to bring the library up-to-date.

The rules are attached to a collection and govern the generation of any library object within that collection, assuming that all its precursors are present. The rules specify how one form can be transformed into another form and what tool performs the transformation. Only the tools listed in the rules are allowed to directly update objects within the catalog. Rules are of the form:

```
        precursor_form -> target_form : tool.
```

The rules are general enough to be able to incorporate foreign language modules, e.g.

```
        pascal_source -> pascal_object_code : pascal_compiler

        pascal_object_code -> ada_object_code :
                               pascal_object_module_converter.
```

An initial form object referenced in the catalog is an object which has no precursors. There is a set of initial form rules which can be used to submit initial forms to the catalog. These rules are of the form:

```
        " " -> initial_form : tool
```

The catalog contains a list of references to objects in the catalog with their ident/form and a unique object id, the most recently generated object of the given ident/form. An object referenced in the catalog is generated only if it is requested and is not up-to-date.

Associated with each object in the library are three precursor lists: the direct precursor list, the initial form precursor list, and the VMM reference list. Each element of a precursor list refers to an object. The object can be referenced either by its collection and ObjId at the time it was added to the list or by its ident/form (if it needs to be added to the list again). An initial form object referenced in a catalog is an object whose precursor lists are null.

An object's direct precursor list contains the ident/form elements which were used directly in order to generate the object. The entries of the direct precursor list contain the information:

(ident/form, collection, ObjId , extra_information)

The collection and ObjId locates which object was used when the element was added to the list. The ident/form is used to check for consistency and refers to the object's ident/form at the time of insertion into the list. The extra_information may include how the ident/form in the direct precursor list is used in relation to the new object(e.g. whether it utilizes the ident/form on the list or whether the new object was transformed from the ident/form on the list). The direct precursor list is used for historical purposes.

The entries in the initial form precursor list contain all the initial form objects which this object depends on; i.e. what was the snapshot of the world it depended upon when this object was generated. This list is used to check whether the catalog is consistent in relation to the object requested. The entries in the initial form precursor list consist of:

(ident/form, collection, ObjId).

The initial form precursor list content is similar to that of the direct precursor list except that the ident/form is the initial form itself. The extra_information is not needed. The ident must be checked so that the initial form in the catalog is the form specified in the initial form precursor list.

After the VMM reference list is built, it contains a complete list of the ident/forms referenced to VMM via cross file pointers and is needed by the object for establishing VMM addressability. The entries in the VMM reference list consist of:

(ident/form, collection, ObjId).

The functions provided by the library object dependency manager are precursor list routines and object consistency routines, which are described below.

3.3.1.2.1 Precursor List routines

Routines are provided to get a precursor list, to obtain an element from a precursor list, and to add an element to a precursor list.

3.3.1.2.1.1 Input

In order to access the precursor lists of an object, the catalog object must be opened. The inputs to the precursor list must specify the object and which precursor list is desired (direct, initial form, or VMM reference). To retrieve a list element one must specify its ident/form. To add an element to a precursor list, the input must also include the collection and ObjId. If the list is a direct precursor list, how the element is used will also need to be recorded when the element is added.

3.3.1.2.1.2 Processing

Normally an object is generated in the catalog with its three precursor lists empty. However, if one object was copied from another object, the new object's initial form precursor list and its VMM reference list are copied from the old object, but the direct precursor list is not. Instead, it contains only the object it was copied from.

As objects are referenced , they are added to the particular new object's precursor list. When a new object is derived from information in another object, the referenced object's ident/form, collection and ObjId are added to the direct precursor list of the new object. Then the initial form precursor list of the referenced object is merged into the new object's initial form precursor list. If the initial form precursor list of the referenced object is empty, then the referenced object itself is added into the new object's initial form precursor list. The VMM reference list of the referenced object is then merged with the VMM reference list of the new object.

3.3.1.2.1.3 Outputs

The precurser list routines obtain an element from a precursor list or add new elements onto a precurser list. When elements are added to the direct precursor list, the elements initial form precursor list may be merged onto the current object's initial form precursor list.

3.3.1.2.2 Library Object Consistency

An object referenced in a catalog is considered up-to-date if its initial form precursors are all still listed as current in their catalogs.

Routines are provided to check whether an object referenced in a catalog is up-to-date and, if not, to try to bring the object up-to-date.

3.3.1.2.2.1 Inputs

An ident/form from a catalog object reference table is checked to see if the ObjId associated with it is up-to-date and if not, optionally bring the ident/form up to date. The catalog and the ident/form must be specified in order to carry out the request.

3.3.1.2.2.2 Processing

B.P An ident/form in a catalog is up-to-date if its initial form precursor list's ObjIds are the same as those in its catalog's object reference table. If the prefix to the ident is in the resource catalog reference list, the ident/form must be the same as those in the corresponding resource catalog. The resource catalog reference list is checked before the catalog reference table since there are usually fewer resource prefixes than idents, thus speeding up lookup. If an ident/form from a resource library is out-of-date, i.e., does not match the one in the resource catalog, a message is issued. Any request to update the object will be ignored if not issued by the owner of the resource catalog, since only the owner may bring the objects referenced in that catalog up-to-date. If any ObjId does not match the catalog's, direct or indirect, then the object is not up-to-date.

In bringing an ident/form from a catalog up-to-date, the object is first checked to see if it is up-to-date, and if it is, nothing else need be done. If the object does not exist or is not up-to-date, the tools mentioned in the set of rules associated with the catalog's collection are used to try to generate an up-to-date ident/form. The most up-to-date ident/form is used as a starting point. If there are any errors in generating new objects following the rules, the catalog is left and marked as in error. The backup catalog (before any objects were changed) can be obtained if requested. Otherwise, the side effect of using the bring up-to-date routine is that the ident/form in the current catalog becomes up-to-date.

3.3.1.2.2.3 Outputs

The object referenced in the catalog is either up-to-date or not and the answer is returned.

A bring-up-to-date call will result either in an object becoming consistent in the catalog, or in the issuing of an error message. Errors can result from the tools used in the rules to generate the object or from a version skew. A version skew happens when a resource is used by two different catalogs through two different paths and each path uses a different version of the resource.

When an error occurs, the catalog is left as it was and a backup may be obtained of reflecting the state of the catalog before object updating was tried. Bringing an object up-to-date may have the side effect of changing some of the ObjIds associated with the ident/forms in the catalog object reference table.

3.3.1.3 Library Object Allocation Package

All library objects in a catalog referenced directly or indirectly, must have non-overlapping VMM segment number ranges. This CPC provides the routines needed to coordinate segment number usage. By default, when a catalog is created, it will be given the lower half of the VMM segment number "address space" (i.e. 0 .. 32767). Catalogs to be used for resource catalogs should instead be created with a VMM segment number range shifted to some part of the upper half of the segment number address space. Note that many catalogs may use the same collection, and hence will automatically have no conflicts over segment numbers (since they are all assigned out of the same pool).

Associated with the collection is a list of segment numbers that can be allocated to any object in its collection. The routines provided in this package handle an object´s use of segment numbers. They can be divided into those that pertain to collection segment numbers and those that pertain to object segment numbers. Allocation of segment numbers proceeds in two steps. First, a range of segment numbers certain to be sufficient is reserved for an object. Second, those numbers actually used are "kept", and cross-indexed with the object, while those numbers found to be unnecessary are returned to the collection for reuse.

3.3.1.3.1 Collection Segment Number

The routines which handle obtaining segment numbers from the collection reserve a set of segment numbers or release a set of segment numbers.

3.3.1.3.1.1 Input

To reserve a set of segment numbers, the collection and the number of segments wanted must be specified. To release unused or no-longer-used segment numbers, the collection and list of segment numbers to be returned must be specified.

3.3.1.3.1.2 Processing

The collection keeps a list of segment numbers which are not used, called the free list. As segments are requested, the segment numbers are removed from the free list. When segments are released, the segment numbers are added back onto the free list.

3.3.1.3.1.3 Output

A list of segment numbers is returned from "reserve". The free list is updated.

3.3.1.3.2 Object Segment Number

When a tool actually allocates a segment number to an object from a pre-reserved segment number list, the segment number is added to the segment number list of the object. The segment number list of the object can be retrieved on request. If an object is deleted, its segment number list will be released.

3.3.1.3.2.1 Input

To allocate a segment number, the segment number and the object must be specified. To release all the segment numbers of the object or to access all the segment numbers of the object, only the object is specified.

3.3.1.3.2.2 Processing

When a segment number is allocated to an object, the segment number is added to the segment number list associated with the object. When the object's segment number list is to be released, the entire list is released to the collection. When a tool requests a segment number list, the list is returned.

3.3.1.3.2.3 Output

The output of obtaining the object's segment number list is the segment number list. The effect of allocation is that a segment number is added to an object's segment number list. The effect of releasing is that the segment numbers associated with the object are now in the general collection pool for use by other objects.

3.3.1.4 Library Configuation Management

This CPC defines primitives to allow tools to create and maintain program libraries that may or may not utilize components residing in other libraries.

The main functions of this CPC are the creation and deletion of collections, the creation and deletion of catalogs, promoting a resource catalog and creating a link to a resource catalog.

3.3.1.4.1 Collection Creation and Deletion

3.3.1.4.1.1 Inputs

The creation or deletion of a collection requires a collection identification. An initial range of VMM segment numbers to be associated with the collection are also required for creation.

3.3.1.4.1.2 Processing

When a collection is created, the CLL and the CAL are initialized to null. The initial range of VMM segment numbers is associated with the collection. After deletion a collection is inaccessible.

3.3.1.4.1.3 Output

The data structures associated with a collection are created (deleted).

3.3.1.4.2 Catalog Creation and Deletion

3.3.1.4.2.1 Inputs

For the creation or deletion of a catalog requires the catalogs ObjId and the identification of the collection it resides in. For creation by copy, the ObjId of the original catalog is also required.

3.3.1.4.2.2 Processing

The object table, the RCR and the prefix index of the catalog are created. For a creation by copy these data structures are copied exactly; otherwise they are set to null.

3.3.1.4.2.3 Output

The data structures associated with the catalog are created.

3.3.1.4.3 Promoting a Resource Catalog

Catalogs may be made accessible to other catalogs as resources by promoting them to the Catalog Access List in their collection.

3.3.1.4.3.1 Inputs

In order to promote a catalog, the identification of the collection and the catalog name must be specified. Also, the catalog must be identified as either an interface of an implementation catalog. In the case of an implementation catalog, its related interface catalog must also be identified.

3.3.1.4.3.2 Processing

When a catalog is promoted, an entry is made to its collection's CAL. For a catalog with no previous revisions, an entry is made in the CAL which includes the name of the catalog, its set of prefixes, the catalog's ObjId, an indication of whether it is an implementation of a resource catalog, and an indication of its related interface/implementation catalog (if one exists). If previous revisions of the catalog exist, the newly promoted revision is assigned a new revision number. If the new revision contains prefixes not contained within other revisions, the prefix list is updated.

3.3.1.4.3.3 Output

An update is made into the CAL of the catalog's collection contain the information specified above.

3.3.1.4.4 Linking to a Resource Catalog

3.3.1.4.4.1 Inputs

In order to create a link from a primary catalog to a resource catalog, the following information must be specified: the primary catalog's name and collection, the resource catalog's name and collection and the revision of the resource catalog (this defaults to the most recent).

3.3.1.4.4.2 Processing

Once it has been established that a link exists between the present collection and the target collection and that the target catalog exists in the target collection's CAL, the prefix index of the target catalog will be checked against the present catalog's prefix index to determine if there are any overlapping prefixes. If there are, an error message will be generated. Otherwise, the target collection identification and catalog name, the revision number of the catalog and its ObjId will be entered into the present catalog's RCR. The entries of the target catalog's RCR will be entered into the present RCR and will be flagged as indirect resource links.

3.3.1.4.4.3 Outputs

The primary catalog's RCR will be updated as described above.

3.3.2 Program Builder

The primary program integration tool is called the Program Builder. It is invoked by a user in order to create an executable program. The Program Builder consists of five components, described in detail below.

3.3.2.1 Builder

The term "building" refers to the processing performed by the Builder on a program library to produce an executable program.

The Builder operates in two phases. The first phase, Program Completion, guarantees that the program is complete and consistent before the second phase, the Linker, binds symbols to their load-time values. The first phase may invoke the Body Generator, to create null bodies for specifications in the program library which have yet to be implemented. It may invoke the Preamble Generator, if the main program requires parameters. Finally, it may cause the invocations of phases of the compiler as a side effect of requests that object modules be brought up-to-date (see the Library Object Dependency Manager CPC).

The second "Linker" phase performs the traditional linking function. It binds all undefined symbols and produces an output load module in the format required by the KAPSE loader.

3.3.2.1.1 Inputs

Inputs to the Builder are the name of a primary catalog and the MAIN ident, which is the name of a "main program" unit, as well as other parameters which are simply passed on to the Linker. The ident must refer to a library unit. If the main unit was a subprogram with parameters, a preamble will be built to process them.

3.3.2.1.2 Processing

The Builder first calls the Program Completeness Checker. For each body found to be missing, the Body Generator is invoked to produce the source for the body, and then a request is made to bring the new body's object module up-to-date, thereby causing the compiler phases to be invoked.

If necessary, the Preamble Generator is invoked, again producing source, this time for the preamble subprogram, which is then compiled as a side effect of a bring-up-to-date request.

Next, an object module with an elaboration CSECT and a map CSECT is created (acting much like an object module for the hypothetical body of package STANDARD). The set of library units accessible from the main unit must be elaborated in the correct order when the program is executed. The elaboration order is computed using the dependency relations of library units, and ELABORATE pragma specifications. The order chosen is recorded in this elaboration CSECT. This CSECT is defined by a list of LSUD nodes, referring via EXTERN reference nodes to either CSECTs or ENTRYs defined in the various object modules making up the program. For all but the last entry, the "selector" portion of the external name identifies either the spec or body elaboration procedure associated with each library unit. The last reference specifies the "body-call" selector for the parameterless main unit which will receive control immediately after library unit elaboration (either the preamble or a user-written parameterless main procedure). The Ada runtime library contains a startup routine which uses the elaboration CSECT to sequence the library unit elaborations. The address of the startup routine is made the initial program counter or "PSW" for the executable program load module.

In addition to the elaboration CSECT, the Builder creates a null statement map "group" CSECT which allows the run time system's debugging support routines to locate the statement tables of each compilation unit, as part of a contiguous CSECT group.

Finally, the Linker is invoked, to combine the object modules and resolve link-time addresses, and produce the executable program load module.

3.3.2.1.3 Outputs

The primary output of the Builder is an executable Ada program. The builder may also have the following side effects:

1. Unit recompilation, if necessary, as a side effect of a bring-up-to-date request.

2. Preamble Generation, which involves the automatic construction of a main program unit when the main unit is a function or a procedure with parameters. The Preamble Generator is invoked to analyze the DIANA of the main subprogram and generate source code for the preamble. The preamble is compiled, creating a unit which now acts as the main unit. When the preamble executes, it converts the string values of parameters from the KAPSE, to the internal representations used in the parameters of the subprogram,

and then invokes the subprogram.

3. Body Generation, which involves the creation of null bodies for referenced specifications whose corresponding bodies have yet to be defined.

4. Elaboration/Map Object Module Creation, which involves the creation of an object module with references to the various library unit elaboration routines, the main unit body, and the group CSECT base for statement tables.

3.3.2.1.4 Special Requirements

3.3.2.2 Preamble Generator

This CPC creates source code for a driver routine for the main subprogram which was written as a function or procedure with parameters. The Preamble Generator may be invoked by the Builder, or by the user directly.

3.3.2.2.1 Inputs

The name of the catalog, and the "Ident" of the main library unit subprogram must be specified.

3.3.2.2.2 Processing

Assuming the specified main unit is named P, the new source for the preamble unit (named DRIVER_P):

1. has a WITH P statement which names the original main unit, as well as all of the units named in its WITH statements (so that subprogram parameter types are visible to the preamble);

2. declares a local variable for each parameter;

3. calls the KAPSE to get the string value for the actual PARAMETERS of the program invocation;

4. for each IN or INOUT parameter, uses the KAPSE function PICK_PARAM to locate either a named parameter-value pair, or the correct positional argument;

5. if a value is found, it is converted to the internal representation for the type using the VALUE attribute of Ada; the value is stored in the corresponding local variable;

6. if the value is not found, a default value is assigned to the local variable; (the normal default mechanism of the compiler cannot be used, since it is not known whether the user will supply a value or not);

7. the main program P is called, passing the local variables as arguments;

8. the inverse conversion is performed (using the IMAGE attribute of Ada) for INOUT or OUT parameters or function value;

9. the KAPSE is called to record the string values of the inverse conversions as the RESULTS of the program invocation.

For each value found in (3) above, if the value is "?" or ":", the preamble writes to standard output the type name expected, along with the enumeration literals if the type is an enumeration, or a range if the type is numeric. This gives a default help capability. If any value is specified in this way, the main program P is not called; instead, a message is written to standard output requesting re-invocation of the program.

The Preamble Generator is invoked by the Program Builder if the main unit is a function or has parameters. The preamble generator may also be invoked explicitly, and the text output may be saved and modified. Thus a standard user interface is ensured for all user-written programs, while, in the presence of special requirements, the normal conventions may easily be overridden.

3.3.2.2.3 Outputs

The Preamble Generator writes the source for the preamble to standard output, with the unit name formed by prepending "DRIVER_" to the name of the specified main subprogram.

3.3.2.2.4 Special Requirements

3.3.2.3 Program Completeness Checker

The Program Completeness Checker identifies all units reachable from a specified main library unit which lack bodies within the program library.

3.3.2.3.1 Inputs

The inputs are the primary catalog and the name of the main library unit.

3.3.2.3.2 Processing

The Program Completeness Checker works in two stages. First, all units accessible from the main library unit are located. Second, for each of these units, any specs without bodies are identified and the names given as library unit/subunit are added to a single output list.

3.3.2.3.3 Outputs

The output is a list of compilation unit names of missing bodies.

3.3.2.3.4 Special Requirements

3.3.2.4 Body Generator

This subprogram creates the source form for a null subprogram, package, or task body corresponding to a given specification, which may be a library unit or a subunit stub.

3.3.2.4.1 Inputs

There are two inputs to this subprogram: a pointer to the DIANA representation of a specification which lacks a body, and the catalog in which that specification resides.

3.3.2.4.2 Processing

Processing the specification depends on the kind of unit being generated.

In the case of a subprogram, the DIANA form of the specification is first converted back to source text. If the subprogram is a subunit, the specification is preceded by the appropriate SEPARATE statement. A null body (BEGIN NULL; END;) is generated in place of the semi-colon at the end of the specification. In the case of a function or a procedure with OUT parameters, the value computed by the null body is undefined.

In the case of a package, a package body skeleton is generated, preceded by a SEPARATE statement if it is a subunit. The DIANA for the package specification is scanned for subprogram and package specifications. For each subprogram specification found, a subprogram body is generated as above. For each package, the routine that is handling the package is recursively invoked. Following the declarations of nested subprogram and package bodies, a null body for the outer package is generated.

In the case of a task body, the null body is generated, without accept statements.

3.3.2.4.3 Outputs

Source for the generated body is written to standard output.

3.3.2.4.4 Special Requirements

3.3.2.5 Linker

3.3.2.5.1 Inputs

The input program library is identified to the Linker through the LIB parameter. This library (specified by the name of its primary catalog) contains one or more compilation units which are to be included in the executable result.

The object module which contains the elaboration CSECT is specified with the ELAB parameter. The external name where execution is to begin is specified by the START parameter.

The name of the "main subprogram" identified in the call to the Builder is specified with the MAIN parameter. The optional CALL parameter is used to specify the filename where the executable program load module should be created. If not specified, the load module is created within the primary catalog, as the "executable" Form under the Ident for the body of the MAIN subprogram.

The OPTIM parameter may be either SPACE or TIME, and determines whether unreference CSECT elimination is performed (only performed if OPTIM=>SPACE).

3.3.2.5.2 Processing

Linker processing occurs in the following stages:

1. optional unreferenced CSECT elimination;

2. CSECT placement;

3. memory image creation.

1. Optional Unreferenced CSECT Elimination. The use of packages and generics in Ada is likely to result in unreferenced subprograms, as are certain large yet rarely referenced attributes (in particular, the "image" table for enumeration types). Since the compiler generates a CSECT for each subprogram body, and other separable constructs such as an "image" table, an unreferenced subprogram or attribute results in an unreferenced CSECT. This processing stage identifies which CSECTs are not referenced by any LREF nodes and excludes them from placement in memory. This

requires a pass over the LREF nodes prior to CSECT placement, and is only executed if the caller has specified OPTIM=>SPACE as a parameter to the Linker (Builder).

2. CSECT Placement. This processing stage assigns relative locations to all of the CSECTs not excluded by the previous stage. Pure and impure CSECTs are grouped in two separate segments. Each CSECT in turn is placed following the previous CSECT in the appropriate segment, with the alignment as specified in the CSECT node.

3. Memory Image Creation. This processing stage allocates pure and impure memory image arrays (VMM variable length arrays) and defines the contents using the CSUD nodes of each CSECT. A storage unit "fill" value is defined for storage units not specified in CSUD nodes or skipped because of CSECT alignment. After all the CSUD nodes for a CSECT have been processed, the LSUD nodes are used to assign the link time values.

As a result of CSECT placement, all components of link-time expression nodes have associated values. Each LSUD node causes a link-time computation to be performed and a value stored into the pure or impure memory image at the location defined in the LSUD node.

3.3.2.5.3 Outputs

There are two outputs from the Linker: an executable program load module, and a relocation map. A human-readable output is available by using the "Link Map/X-Ref Lister" (a CPC of PLTOOLS).

The load module is an attributed database object, including the following information:

1. The pure and impure memory images, with a start address.

2. The relocation map as an attribute, recording the placement of each global CSECT in the program.

3. A window attribute providing access back to the primary catalog of the program library which provides the debugger or any other program analyzer with access to the library units which generated the program.

The relocation map is a VMM file containing a mapping set from external symbol name (i.e. VMM locator plus selector), to the value determined by the Linker, and the length if the symbol refers to a CSECT.

3.3.2.5.4 <u>Special Requirements</u>

3.3.3 Program Library Tools

The Program Library Tools are a set of Ada programs which can manipulate, display, or extract information from the program library, plus tools to aid in the minimization of re-compilation costs.

3.3.3.1 Program Library Manager

Program Library Manager: This program provides access for the interactive user to the various primitives of the Program Library Interface Packages. It allows the user to create, manipulate, display, and delete the various objects forming a program library.

3.3.3.1.1 Inputs and outputs

The inputs include a specification of the catalog or collection, plus an optional specification of the library unit of interest (as Ident/Form). The input must specify the operation, either create, copy, link, delete, or display.

In the case of display, the input must select the various attributes of the specified collection (CLL or CAL), catalog (Object Reference "able or RCR), or unit (Precursor lists, Size, Date created, whether Up-to-date, etc.) for display.

3.3.3.1.2 Processing

The processing for this CPC consists entirely of invoking the primitives available within the Program Library Interface Packages (PIF.PLIF), and then if appropriate, formatting the output for display.

3.3.3.1.3 Outputs

For all but the display operations, the outputs are simply an update collection/catalog/unit.

For display operations, no changes are made to the program library components, but rather the information requested is displayed on the standard output.

3.3.3.1.4 Special Requirements

3.3.3.2 Change Analyzer

This program takes two revisions of a unit (U and U´) and compares the DIANA trees.

3.3.3.2.1 Inputs

Two DIANA trees, U and U´.

3.3.3.2.2 Processing

The DIANA tree of U is walked in a recursive algorithm which visits each construct. Two techniques for comparision are used.

1. If the DIANA construct in U is the defining occurrence of an identifier, the same identifier is looked up by name in the corresponding scope of U´.

2. Otherwise, a structural tree comparison is performed, stopping at the first unequal comparision.

The output of the comparison is a VMM subdomain that contains a mapping set. Each element of the set is a comparision node, with its membership testing criterion being the VMM locator of the node in U. The comparison node contains the VMM locator of the corresponding node in U´. If the comparison node represents the defining occurrence of an identifier, the comparison node records the result of the comparison (equal, not equal, not found). Nodes compared as a result of structural tree comparison are in the set only if they compared equal. Nodes are considered equal if all semantic and storage allocation "attributes" are equal.

After the mapping set is built, an inverse set is constructed with the membership criterion being the VMM locator of the node in U´. Then a structural walk of the DIANA tree of U´ is performed, checking that each node representing a defining occurrence of an identifier is in the inverse set. If not, a new member of the set is created indicating that the corresponding node in U is not found. (Thus adding a new identifier in U´ is properly represented in the inverse set.)

The mapping sets are used by the MAP program to determine which units need recompilation.

3.3.3.2.3 Outputs

A VMM subdomain is produced with the forward and reverse mapping sets, a flag indicating whether all comparisons succeeded, and a flag indicating whether every pair of VMM locators in U and U′ have identical subdomain offsets. The last flag allows the MAP program to optimize the mapping by instructing the VMM system to substitute only the subdomain number part of a VMM locator.

The program optionally produces a human-readable output showing the differences found by the comparison.

3.3.3.2.4 Special Requirements

3.3.3.3 Recompilation Minimizer (MAP)

The user invokes this program to minimize the impact of the submittal of an updated unit to a catalog. By so doing, the user is asserting that the new unit is nearly identical, or at least similar enough so that some or all of the referencing units do not need recompilation.

3.3.3.3.1 Inputs

A catalog, and the Ident/Form of the new unit recently submitted.

3.3.3.3.2 Processing

Assuming the backup object for the given Ident/DIANA is "U," and the new Ident/DIANA is "U´," the Change Analyzer program is invoked to produce the mapping sets between the two units, U and U´. Each unit in the library is inspected to see if it refers to unit U. If it does, the global cross reference set in the unit is inspected for individual references to identifiers in unit U. For each identifier used, the mapping set produced by the Change Analyzer program is checked to see if the identifier is identical in U´. If all identifiers used in U´ are found to be identical U´, the unit does not need recompilation. Note that if U is a package and the referencing unit has a USES statement, U and U´ must be identical for all identifiers, not just the referenced ones.

If any used identifers are found to be changed or deleted in U´, the unit must be recompiled. If not, the unit´s precursor lists are adjusted to refer to the new unit, along with the mapping set.

3.3.3.3.3 Outputs

After all referencing units have been checked, if any have created references to the mapping set built by the Change Analyzer, it is saved as a VMM locator association set. The VMM system will then map any references to nodes in U with the identical nodes in U´.

3.3.3.3.4 Special Requirements

3.3.3.4 Link Map/X-Ref Lister

Link Map/X-Ref Lister produces a human readable link map, including linker error messages (if any) and global cross reference information.

3.3.3.4.1 Inputs

The primary inputs are the name of the catalog, and the Ident/Form of the link map. The caller may specify listing options:

1. UNITS -- All CSECTs for unit specs and bodies.
2. GLOBALS -- All global CSECTs/ENTRYs
3. ALL -- All local or global CSECTs/ENTRYs.
4. XREF -- Cross reference for symbols listed.

3.3.3.4.2 Processing

The relocation map is scanned for symbols satisfying the listing options (UNITS is the default). The DIANA in which the definition occurs, as determined by the VMM reference list of the map, provides the necessary human-readable representation of the symbol and its value.

If a cross-reference is desired, the output is stored on a file, and then sorted by a second phase of processing before being output.

3.3.3.4.3 Outputs

The listing is produced on the standard output.

3.3.3.4.4 Special Requirements

3.3.3.5 Source Reconstructor

This program extracts a text file associated with a compilation unit from a program library.

3.3.3.5.1 Inputs

The input includes the name of the catalog, and the ident/form of the compilation unit from which the source is to be reconstructed. Either an AST form or a DIANA form may be specified (AST is the default). An optional parameter may specify verbatim reconstruction, or a reformatted reconstruction with a standardized indenting and commenting convention.

3.3.3.5.2 Processing

The AST or the DIANA, as specified, is walked, following the abstract structure attributes and the lexical attributes of each node, producing the expanded output on standard output. When walking DIANA, it may be necessary to remove certain canonicalizations.

3.3.3.5.3 Outputs

The formatted listing is written to standard output.

3.3.3.6 Unit Lister

The Unit Lister produces listings of the source text, symbol table attributes, cross reference and assembly code for the compiler. The COMPILE request may be specified with LIST option. Alternatively, because all the information that the lister needs to generate a listing (except for the assembly listing) has already been permanently saved, the compiler may be run with the LIST option turned off and at a later time the Unit Lister may be run separately to produce a listing. The user's request to produce listing is represented as:

```
    LISTER UNIT=>unit_name LIB=>lib_name [options]
```

3.3.3.6.1 Inputs

Inputs to the Unit Lister are the library, the name of the unit to be listed, and a set of options.

3.3.3.6.2 Processing

The compiler invokes the Unit Lister if the LIST option is specified in the COMPILE request. The Unit Lister gets LIST values from the string that is saved as OPTIONS attribute of the compilation unit. When Unit Lister is called by a user, values are taken from the user's specified option list. The options to the lister and their values are specified below.

```
   LIST=> {ON, OFF, SOURCE, NOSOURCE, ATTRS, NOATTRS, XREF, NOXREF,
           ASSEMBLY, NOASSEMBLY}
```

SOURCE, NO-values and OFF are default.
ON produces a full listing with all Unit Lister options present.
OFF produces no listing; a listing may be requested at a later time.

SOURCE controls listing of the source text. The source text is reconstructed from information stored in the lexical attributes of the DIANA. The DIANA is also referenced to generate the nesting counts, the statement numbers, the name of the current scope and the line cross reference.

ATTRS controls listing of symbol table attributes of identifiers. The symbol table is a permanent DIANA data structure for each compilation unit and as such is stored in the library. The symbol table in the library is the name table augmented with the DIANA DEF_ID nodes that are the corresponding definitions for the identifier. The name table defines a mapping between identifiers and DIANA DEF_ID nodes contained within the DIANA tree.

XREF controls listing of cross-reference for all identifiers. The LIST XREF option causes Statinfo, during the Middle Phase of the Compiler, to collect cross reference information and save it in DIANA. The cross reference information is then used to locate all of the compilation unit's uses of DEF_ID nodes. Many of the symbol table attributes are also listed in the cross reference.

ASSEMBLY controls listing of generated code. Assembly listing lines are generated during the FINAL phase of the Back End of Compiler.

3.3.3.6.3 Outputs

The output of the Unit Lister depends upon the values of the LIST option. See Section 3.2.4.7 for the description of the Unit Lister Output Format.

3.3.3.7 Foreign Object Module Importer

The Foreign Object Module Importer converts non-standard object modules formats to the PIF Object Module Format.

Flexibility is provided by the general rule-based library definition to incorporate foreign languages and their transformation within the context of the PIF.

3.3.3.7.1 Inputs

The inputs required are the name of the catalog, the name of the foreign object module, the "form" of the foreign object module, and the Ident of an Ada spec or stub already in the catalog whose body is to be implemented using the foreign object module.

3.3.3.7.2 Processing

Each distinct type of foreign object module may require a separate tool to do the conversion. This tool must appear in the rules list associated with the catalog's collection, as a tool which can transform some foreign object module form into the standard object module form. The general Foreign Object Module Importer simply copies the specified foreign object module into the library under the "form" name supplied as argument (which it must be allowed to do in the rules list), and then uses the general bring-up-to-date function to run the object module importer specific to the foreign form provided.

3.3.3.7.3 Outputs

The catalog is updated to include the foreign object module, and its conversion into standard object module form.

3.4 Adaptation

3.5 Capacity

4. Quality Assurance

(To be specified)